Preemptible I/O Scheduling of Garbage Collection for Solid State Drives

Junghee Lee*, Youngjae Kim†§, Galen M. Shipman†, Sarp Oral†, and Jongman Kim*

***Abstract*—Unlike hard disks, flash devices use out-of-update operations and they require a garbage collection (GC) process to reclaim invalid pages to create free blocks. This GC process is a major cause of performance degradation when running concurrently with other I/O operations as internal bandwidth is consumed to reclaim these invalid pages. The invocation of the GC process is generally governed by a low watermark on free blocks and other internal device metrics that different workloads meet at different intervals. This results in I/O performance that is highly dependent on workload characteristics. In this paper, we examine the GC process and propose a semi-preemptible GC scheme that allows GC processing to be preempted while pending I/O requests in the queue are serviced. Moreover, we further enhance flash performance by pipelining internal GC operations and merge them with pending I/O requests whenever possible. Our experimental evaluation of this semi-preemptible GC scheme with realistic workloads demonstrate both improved performance and reduced performance variability. Write-dominant workloads show up to a 66.56% improvement in average response time with a 83.30% reduced variance in response time compared to the non-preemptible GC scheme. In addition, we explore opportunities of a new NAND flash device that supports suspend/resume commands for read, write and erase operations for fully preemptible GC. Our experiments with a fully preemptible GC enabled flash device show that request response time can be improved by up to 14.57% compared to semi-preemptible GC.**

***Index Terms*—Solid-state Drives (SSDs), Garbage Collection, Preemptive I/O, I/O Scheduling, Flash Memory, Storage Systems.**

I. Introduction

HARD disk drives (HDD) are the primary storage media for large-scale storage systems and have been for a few decades. Recently, NAND flash memory based solid-state drives (SSD) have become more prevalent in the storage marketplace with advancements in the semi-conductor technology. Unlike HDDs, SSDs do not have mechanically moving parts. SSDs offer several advantages over HDDs such as lower access latency, higher resilience to external shock and vibration, and lower power consumption which results in lower operating temperatures. Other benefits include lighter weight and flexible designs in terms of device packaging. Moreover, recent reductions in cost (in terms of dollar per GB) have accelerated the adoption of SSDs in a wide range of application areas from consumer electronic devices to enterprise-scale storage systems.

One interesting feature of flash technology is the restriction of write locations. The target address for a write operation should be empty [1], [15]. When the target address is not empty the invalid contents must be erased for the write operation to succeed. Erase operations in NAND flash are nearly an order of magnitude slower than write operations. Therefore, flash-based SSDs use out-of-place writes unlike in-place writes on HDDs. To reclaim stale pages and to create space for writes, SSDs use a Garbage Collection (GC) process. The GC process is a time-consuming task since it copies non-stale pages in blocks into the free storage pool and then erases the blocks that do not store valid data. A block erase operation takes approximately 1-2 milliseconds [1]. Considering that valid pages in the victim blocks (to be erased) need to be copied and then erased, GC overhead can be quite significant.

GC can be executed when there is sufficient idle time (i.e., no incoming I/O requests to SSDs) with no impact to device performance. Unfortunately, prediction of idle times in I/O workloads is challenging and some workloads may not have sufficiently long idle times. In a number of workloads incoming requests may be bursty and an idle time can not be effectively predicted. Under this scenario the queue-waiting time of incoming requests will increase. Server-centric enterprise data center and high-performance computing (HPC) environment workloads often have bursts of requests with low inter-arrival time [22], [15]. Examples of enterprise workloads that exhibit this behavior include on-line-transaction processing applications, such as OLTP and OLAP [6], [24]. Furthermore, it has been found that HPC file systems are stressed with write requests of frequent and periodic checkpointing and journaling operations [31]. In our study of HPC I/O workload characterization at Oak Ridge Leadership Computing Facility (OLCF), we observed that the bandwidth distributions are heavily long-tailed and write requests occupy more than 50% of workloads [22].

In this paper, we propose a semi-preemptible garbage collection scheme (PGC) that enables the SSDs to provide sustainable bandwidths in the presence of these heavily bursty and write-dominant workloads. We show that the PGC can achieve higher bandwidth over the non-preemptible GC scheme by allowing preemption of an on-going GC process to service incoming requests. While our previous work [26] discusses only semi-preemptible GC, this paper also demonstrates the feasibility of fully-preemptible GC (F-PGC) that supports

*J. Lee and *J. Kim are with the School of Electrical and Computer Engineering, Georgia Institute of Technology, Atlanta, GA, 30332 USA e-mail: {jlee36, jkim}@ece.gatech.edu.

†Y. Kim, †G. Shipman, and †S. Oral are with Oak Ridge National Laboratory, Oak Ridge, TN 37831 USA e-mail: {kimy1, gshipman, oralhs}@ornl.gov. §Y. Kim is a corresponding author.

suspend/resume commands for read, write and erase operations.

This paper makes the following contributions:

- We empirically observe the GC related performance degradation on commercially-off-the-shelf (COTS) SSDs for bursty write-dominant workloads. Based on our observations, we propose a novel semi-preemptible GC scheme for SSDs.
- We identify preemption points that can minimize the preemption overhead. We use a state diagram to define each state and state transitions that result in preemption points. For experimentation we enhance the existing Microsoft Research (MSR)'s SSD simulator [1] to support our PGC algorithm. We show an improvement of up to 66.56% in average response time for overall realistic applications.
- We investigate further I/O optimizations to enhance the performance of SSDs with PGC by merging incoming I/O requests with internal GC I/O requests and pipelining these resulting merged requests. The idea behind this technique is to merge internal GC I/O operations with I/O operations pending in the queue. The pipelining technique inserts the incoming requests into GC operations to reduce the performance impact of the GC process. Using these techniques we can further improve the performance of SSDs with PGC enabled by up to 13.69% for the Cello benchmark.
- We conduct a comprehensive study with synthetic traces by varying I/O patterns (such as request size, inter-arrival times, sequentiality of consecutive requests, read and write ratio, etc.) We present results of a realistic study with enterprise-scale server and HPC workloads. Our evaluations with PGC enabled SSD demonstrate up to a 66.56% improvement in average I/O response time and an 83.30% reduction in response time variability.
- We discuss the feasibility of F-PGC. When the suspend/resume commands are only allowed for the erase operation, the average response time is improved by up to 8.00% compared to PGC. When they are supported or read, write, and erase operations, the average response time is improved by up to 14.57%.

II. Background and Motivation

Unlike rotating media (HDD) and volatile memories (DRAM) which only need read and write operations, flash memory-based storage devices require an erase operation [29]. Erase operations are performed at the granularity of a block which is composed of multiple pages. A page is the granularity at which reads and writes are performed. Each page on flash can be in one of three different states: (i) *valid*, (ii) *invalid* and (iii) *free/erased*. When no data has been written to a page, it is in the erased state. A write can be done only to an erased page, changing its state to valid. Erase operations (on average 1-2 ms) are significantly slower than reads or writes. Therefore, out-of-place writes (as opposed to in-place writes in HDDs) are performed to existing free pages along with marking the page storing the previous version invalid. Additionally, write latency can be higher than the read latency by up to a factor 10. The lifetime of flash memory is limited by the number of erase operations on its cells. Each memory cell typically has a lifetime of 10^3-10^9 erase operations [14]. *Wear-leveling* techniques are used to delay the wear-out of the first flash block by spreading erases evenly across the blocks [19], [8].

Flash-based SSD provides a host interface (such as Fiber-Channel, SATA, PATA, and SCSI) to appear as a block I/O device to the host computer [26]. The main controller is composed of two units, the processing unit (such as an ARM7 processor) and fast working memory (such as SRAM or DRAM). The virtual-to-physical mappings are processed by the processor and the data-structures related to the mapping table are stored in working memory in the main controller. The software module related to this mapping process is called the Flash Translation Layer (FTL). A part of working memory can be also used for caching data.

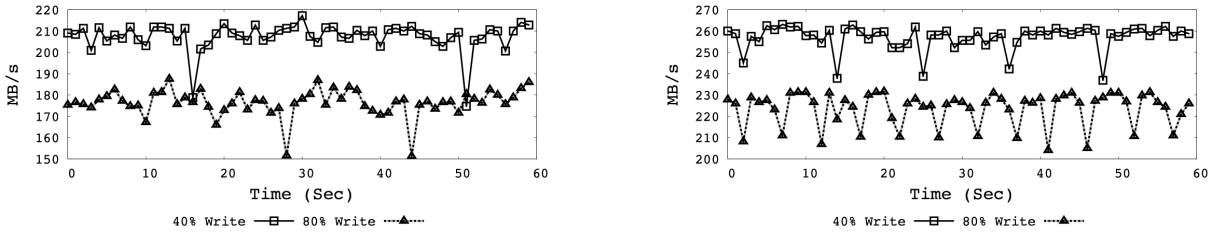
A storage pool in an SSD is composed of multiple flash memory planes. The planes are implemented in multiple dies. For example, the Samsung 4 GB flash memory has two dies. A die is composed of four planes, each of size 512 MB [1]. A plane consists of a set of blocks. The block size can vary (64KB, 128KB, 256KB, etc.) depending on the memory manufacturer. The SSD can be implemented using multiple planes. SSD performance can be enhanced by interleaving requests across the planes, which is achieved by a multiplexer and demultiplexer between working memory and flash memories [1].

The Flash Translation Layer (FTL) is a software layer that translates logical addresses from the file system into physical addresses on a flash device. The FTL helps in emulating flash as a normal block device by performing out-of-place updates thereby hiding the erase operations in flash. The FTL mapping table is stored in a small, fast working memory. FTLs can be implemented at different granularities in terms of the size of a single entry capturing and address space in the mapping table. Many FTL schemes [11], [27], [20], [28] and their improvement by write-buffering [21] have been studied. A recent page-based FTL scheme called DFTL [15] utilizes temporal locality in workloads to overcome the shortcomings of the regular page-based scheme by storing only a subset of mappings (those likely to be accessed) on the limited working memory and storing the remainder on the flash device itself.

Due to out-of-place updates, flash devices must clean stale data for providing free space (similar to a log-structured file system [35]). This cleaning process is known as garbage collection (GC). During an ongoing GC process incoming requests are delayed until the completion of the GC when their target is the same flash chip that is busy with GC. Current generation SSDs use a variety of different algorithms and policies for GC that are vendor specific. It has been empirically observed that GC activity is directly correlated with the frequency of write operations, amount of data written, and/or the free space on the SSD [9]. The GC process can significantly impede both read and write performance, increasing queuing delay.

A. Motivation

In order to empirically observe the effect of GC on the service times of incoming I/O requests, we conducted block-



(a) Write-dominant (80% write) (b) Read-dominant (20% write)

Fig. 1: Bandwidth variability comparison for MLC and SSD SSDs for different write percentages of workloads.

level I/O performance tests with various SSDs. Table I shows their detail specifications. We selected the Super Talent 128 GB SSD [38] as a representative of multi-level cell (MLC) SSDs and the Intel 64 GB SSD [18] as a representative of single-level cell (SLC) SSDs. We denote the SuperTalent MLC, and Intel SLC devices as SSD(A), and SSD(B) in the remainder of this study, respectively. All experiments were performed on a single server with 24 GB of RAM and an Intel Xeon Quad Core 2.93GHz CPU [17], running Linux (Lustre-patched 2.6.18-128 kernel). The *noop* I/O scheduler with FIFO queuing was used [33].

TABLE I: Characteristics of SSDs used in our experiments.

Label	SSD(A)	SSD(B)
Company	Super-Talent	Intel
Model	FTM28GX25H	SSDSA2SH064G101
Type	MLC	SLC
Interface	SATA-II	SATA-II
Capacity (GB)	120	64
Erase (#)	10-100K	100K-1M

To measure the I/O performance we use a benchmark that exploits the *libaio* asynchronous I/O library on Linux. Libaio provides an interface that can submit one or more I/O requests in one system call *iosubmit()* without waiting for I/O completion. It can also perform reads and writes on raw block devices. We used the direct I/O interface to bypass the I/O buffer cache of the OS by setting the *O-DIRECT* and *O-SYNC* flags in the file *open()* call.

We experimented with two workloads of 40% and 80% writes. The I/O request size was fixed at 512KB, and request access patterns were completely random. We measured bandwidth every second. Figure 1(a)&(b) show time-series plots of our bandwidth measurements for SSD(A)&(B). We observe that (i) several bandwidth drops occur over time for all experiments, and (ii) the bandwidth drops are more frequent for the workloads with a higher amount of writes. In order to fairly compare the bandwidth variability for different workloads, we calculated coefficient of variation (CV)[1] values for each experiment.

TABLE II: Average, Standard Deviation, and CV values for Figure 1(a)&(b).

Type	Metric	Write (%) in Workload	
		80	40
SSD(A)	(avg, stddev)	(176.4, 6.37)	(207.4, 6.73)
	CV	0.03599	0.03249
SSD(B)	(avg, stddev)	(223.5, 7.96)	(257.1, 5.86)
	CV	0.03561	0.02279

[1]Coefficient of variation (C_v) is a normalized measure of dispersion of a probability distribution, that is, $C_v = \frac{\sigma}{\mu}$.

Table II compares the CV values for the experiments. We see that a higher write percentage in the workload shows higher CV values, which means higher bandwidth variability. We suspect that this performance variability is attributable to the GC process. This insight led to our design and development of a preemptible garbage collector. The basic idea of the proposed technique is to service an incoming request even while GC is running.

III. Preemptible Garbage Collection

A. Semi-Preemptible GC

Figure 2 shows a typical garbage collection process. Once a victim block is selected during GC, all the valid pages in that block are moved into an empty block and the victim block is erased. A moving operation of a valid page can be broken down to *page read*, *data transfer*, *page write*, and *meta data update* operations. If both the victim and the empty block are in the same plane, the data transfer operation can be omitted by using a copy-back operation [1] if the flash device support this operation.

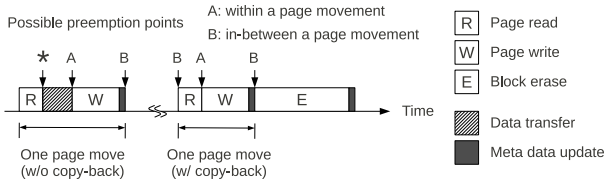


Fig. 2: Description of operation sequence during GC.

We identify two possible preemption points in the GC sequence marked as 'A' and 'B' in Figure 2. Preemption point 'A' is within a page movement and 'B' is in-between page movement. Preemption point 'A' is just before a page is written and 'B' is just before a new page movement begins. We may also allow preemption at the point marked with a (*), but the resulting operations are the same as those of 'A' as long as the preemption during data transfer stage is not allowed. At preemption point 'A', only a write request can be serviced if the NAND flash memory supports pipelining commands of the same type because the page buffers are already occupied by the previous read page operation. The pipelining will be described in more detail in Section III-C. If the NAND flash does not support pipelining, no request can be serviced at preemption point 'A'. In contrast, preemption point 'B' can service any kind of incoming request.

Figure 3 illustrates our proposed semi-preemption scheme. The subscripts of *R* and *W* indicate the page number accessed.

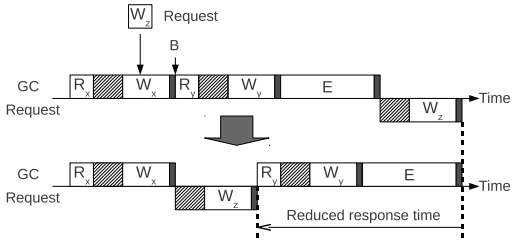


Fig. 3: A semi-preemption. R, W, and E denote read, write, and erase operations, respectively. The subscripts indicate the page number accessed.

Suppose that a write request on page *z* arrives while writing page *x* during GC. With a conventional non-preemptible GC, the request should be serviced after GC is finished, as illustrated in the upper diagram of Figure 3. If GC is fully preemptible, the incoming request may be serviced immediately. To do so, the on-going writing process on *x* should be canceled or suspended first. However, there is no NAND flash memory so far that allows on-going read/write operations to be canceled or suspended, to our best knowledge. The fully preemptible GC is discussed in more detail in Section IV. In PGC, the preemption occurs only at preemption points. As shown in the bottom of Figure 3, the incoming request on page *z* is inserted at preemption point 'B'. As a result, the response time of writing page *z* is substantially reduced.

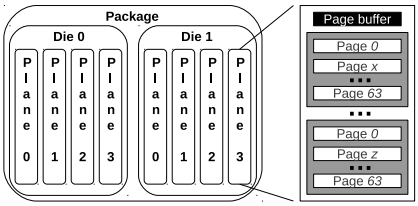


Fig. 4: The internal structure of NAND flash device.

1) Space Overhead Discussion: Our proposed semi-preemption does not require an additional buffer to service incoming requests while GC is running because it exploits the page buffer that already exists in the flash device. Figure 4 shows the internal structure of a typical NAND flash device. One device consists of multiple dies, each of which contains multiple planes. Each plane has a page buffer and number of blocks. The pages in the block cannot be directly accessed. To read data from a page, the data should be copied to the page buffer and read from that page buffer. Data should be written through the page buffer in a similar manner.

To move page *x* in GC, the data on page *x* should be copied to the page buffer in the plane where page *x* is located. Then, the data should be moved to a page buffer where a free block is located, and then written onto a page in the free block. At preemption point 'B' the page buffers are available in both planes. Therefore, to service read and write requests on any page, the service can be launched through the page buffer. In contrast, at preemption point 'A' the page buffer is already occupied by the data of page *x*. If the incoming request is on the same plane as *x*, it cannot be serviced because the page buffer is not available. Only if the flash device supports pipelining, and the incoming request is a write request, the request can be serviced. For example, data of the incoming write request can be written to the page buffer while data in the page buffers are being written to a page in the free block.

2) Computation Overhead Discussion: Our proposed semi-preemption does not require an interrupt. Due to the small number of preemption points it can be implemented by a polling mechanism. At every preemption point, the GC process looks up the request queue. This may involve a function call, a small number of memory accesses to look up the queue, and a small number of conditional branches. Assuming 20 instructions and 5 memory access per looking up, $10ns$ per instruction ($100MHz$), $80ns$ per memory access, the look-up operation takes $600ns$. One page move involves at least one page read which takes $25\mu s$ and one page write which takes $200\mu s$ [1]. Since there are two preemption points per one page move, the overhead of looking up the queue per one page move can be estimated as $1.2\mu s/225\mu s = 0.53\%$.

To resume GC after servicing the incoming request, the context of GC needs to be stored. The context to be stored at preemption points 'A' and 'B' is very small because it doesn't require an additional buffer to service the incoming requests. At preemption point 'A', the block number of the victim block and the page number of the page stored in the page buffer need to be stored in the working memory. At preemption point 'B', only the block number of the victim block needs to be stored. Because the meta data is already updated, the incoming request can be serviced based on the mapping information. Thus, the memory overhead for PGC is negligible.

B. Merging Incoming Requests into GC

While servicing incoming requests during GC, we can optimize the performance even further. If the incoming request happens to access the same page in which the GC process is attending, it can be merged.

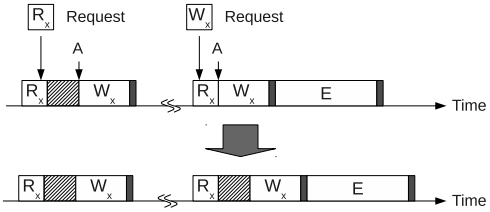


Fig. 5: Merging an incoming request to GC.

Figure 5 illustrates a situation where the incoming request of a read or write on page *x* arrives while page *x* is being read by the read stage of GC. The read request can be directly serviced from the page buffers and the write request can be merged by updating data in the page buffers. In case of copy-back operations, the data transfer is omitted, but to exploit merging, it cannot be omitted. As for the read request, data in the page buffer should be transferred to service the read request. For the write request, the requested data should be written to the page buffer. We can increase changes of I/O merging operations by re-ordering the sequence of pages to be moved from the victim block. Suppose page *x* moves and *y* and *z* then, move. During GC, the order of pages to be moved does not matter. Thus, when a request on page *z* arrives, it can be reordered as *z*, *x*, and *y*.

C. Pipelining Incoming Requests with GC

The response time can be further reduced even if the incoming request is on a different page from valid pages in the victim block to be moved. To achieve this we take advantage of the internal parallelism of the flash device. Depending on the type of the flash device, internal parallelism and its associated operations can be different. In this paper, we consider pipelining [32] as an example. Pipelining allows overlapping the data transfer and the write operations as illustrated at the bottom of Figure 6. If two consecutive requests are of the same type, i.e., read after read, or write after write, these two requests can be pipelined.

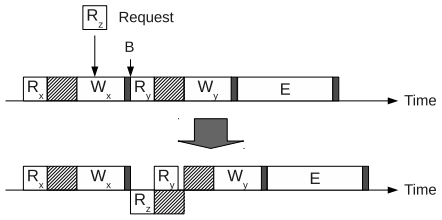


Fig. 6: Pipelining an incoming request with GC.

Figure 6 illustrates a case where an incoming request is pipelined with GC. As an example, lets assume that there is a pending read operation on page z at the preemption point 'B' where a page read on page y is about to begin. Since both operations are read, they can be pipelined. However, if the incoming request is a write operation, they can not be pipelined at preemption point 'B' as two operations need to be issued at 'B' and they are not of the same type. In this case, the incoming request should be inserted serially as shown in Figure 3.

It should be noted that pipelining is only an example of exploiting the parallelism of an SSD. An SSD has multiple packages, where each package has multiple dies, and each die has multiple planes. Thus, there are various opportunities to insert an incoming requests into GC as means of exploiting parallelism at different levels. We may interleave servicing requests and moving pages of GC in multiple packages or issue a multi-plane command on multiple planes [32]. According to the GC scheme and the type of operations the flash device supports, there are many instances of exploiting parallelism.

D. Level of Allowed Preemption

The drawback of preempting GC is that the completion time can be delayed which may incur a lack of free blocks. If the incoming request does not consume free blocks, it can be serviced without depleting the free block pool. However, there may be a case where the incoming request is a write request whose priority is high but there are not enough free blocks. The incoming requests may be prioritized by the upper-layer file system. In such a case, GC should be finished as soon as possible.

Based on these observations, we identify four states of GC:

- **State 0 (S_0):** GC execution is not allowed.
- **State 1 (S_1):** GC can be executed but all incoming requests are allowed.
- **State 2 (S_2):** GC can be executed but all free block consuming incoming requests are prohibited.
- **State 3 (S_3):** GC can be executed but all incoming requests are prohibited.

Conventional non-preemptible GC has only two states: 0 and 3. Generally, switching from S_0 to S_3 is triggered by threshold or idle time detection. Once the number of free blocks falls below a pre-defined threshold the state is changed from S_0 to S_1 and from S_1 to S_2. We call the conventional non-preemptible threshold as *soft* but in our proposed design the system allows for the number of free blocks to fall below the soft threshold. We define a new threshold called *hard* which prevents a system crash by running out of free blocks. Switching from S_2 to S_3 is triggered by the type of incoming requests. If the incoming request is write whose priority is high, it switches to S_3. The priority should depend on requirements of the system.

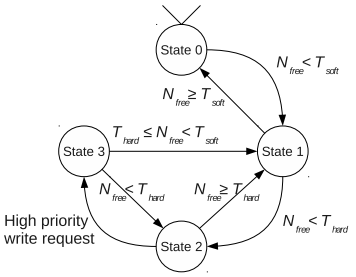


Fig. 7: State diagram of semi-preemptible GC.

Figure 7 illustrates the state diagram. If the number of free blocks (N_{free}) becomes less than the soft threshold (T_{soft}), the state is changed from 0 to 1. If the free block pool is recovered and N_{free} is larger than T_{soft}, then the system switches back to state 0. If N_{free} is less than the hard threshold (T_{hard}), the system switches to S_2 or remains in S_1. In state 2, the system will move to S_1 if N_{free} is larger than T_{hard}. If there is an incoming request whose priority is high, the system switches to S_3. While in S_3, after completing current GC and servicing the high priority request, the system will switch to S_1 or S_2 according to N_{free}.

IV. Fully-Preemptible GC

In Section III, we have presented a novel semi-preemptible garbage collector with several I/O scheduling algorithms. In this section, we present a fully-preemptible GC mechanism by allowing preemption on any on-going I/O operations.

A. Fully-Preemptible GC (F-PGC)

A typical NAND flash accesses the NAND flash cells through a page buffer. If a read command is issued, the requested page is copied from the NAND flash cell to the page buffer and the requester reads data from the page buffer. Similarly, to write data to the NAND flash memory, the requester writes data to the page buffer and issues a write command. These commands are used as atomic operations, i.e., if the commands are issued, they cannot be suspended or canceled until they finish. However, the physical operations on NAND flash cells are not atomic. Current implementation of flash operations, such as page read, page write and block

erase, have been implemented atomic because the NAND flash interface [30] doesn't support preemption, however, they can be implemented preemptible. We add a *suspend* command and a *resume* command to the interface to implement fully-preemptible GC (F-PGC). AMD's NAND flash memories [37] used to support suspend/resume commands for the erase operation. The suspend and resume commands should be operable with read and write operations in addition to the erase operation to support fully preemptible garbage collection.

B. Design for Suspend and Resume Commands

The flash operations can be broken-down into multiple phases. Just like the semi-preemption of the GC process, the flash operations can be preempted in-between phases. For example, the NAND flash memory usually employs the incremental step pulse programming (ISPP) as its write and erase method because it offers fast write/erase performance coping with process variations [3]. It tries to write/erase by a pulse with an initial voltage e.g. 15V and then verifies if it is successful. If not, it keeps increasing the voltage by a step e.g. 0.5V until it succeeds. Therefore, the write/erase operation consists of repeated pulse and verify phases. In-between phases, it is possible for the operation to be suspended. The suspend command forces the on-going command to stop its operation until the resume command restarts its operation. While a previously issued command is suspended, a new command may be issued unless the new command is on the same page or block that is occupied by the suspend command.

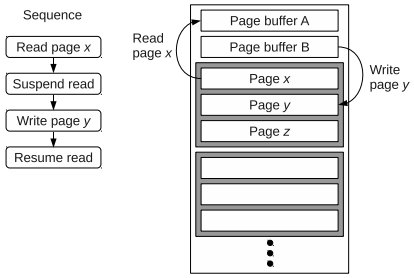


Fig. 8: An example of preempting an on-going flash operation with the suspend command.

Figure 8 gives an example of using suspend/resume commands. For implementing the states of suspension and resumption, an extra page buffer is required. Suppose that a read command is issued on page x. The data in page x is copied to page buffer A. Before the read command finishes, we may issue a suspend command. While the read command is suspended, one can issue a write command on page y. The page y should be different from page x but it can be in the same block of page x. However, if the suspended command is the erase operation, the new command cannot be on any page in that block. The data to be written to page y should be stored in page buffer B. Once the write command finishes, the previous read command that was suspended can resume. Two commands can never suspend at the same time. In this example, write operation can never suspend while read command is suspended. At the cost of additional page buffers, we can allow more commands to be suspended at the same time. However, in order to implement F-PGC, suspending only one command at a time is enough.

If the flash device supports suspend and resume commands but has only one page buffer per plane, servicing incoming requests could be limited according to the availability of the page buffer. For the above-mentioned example, when the on-going read command is suspended, its page buffer is partially occupied. If the incoming write request is on a different plane, it can be serviced immediately, but if it is on the same plane, it should wait until the on-going read command finishes because the page buffer is not available for servicing the request.

After issuing a command, FTL should check if the command is completed either by polling the status register or by receiving an interrupt. Servicing an interrupt incurs non-negligible overhead because of mode switching. For example, ARM1176 needs 200 cycles per switch and Cortex-A8 needs 1200 cycles per switch [2]. Since checking by an interrupt incurs non-negligible mode switching overhead to implement F-PGC, a polling mechanism has been implemented.

C. Operation Sequence

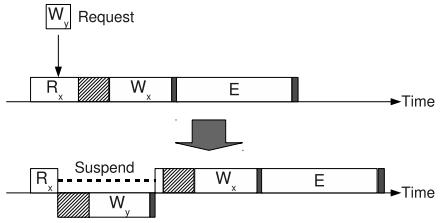


Fig. 9: Operation sequence of fully preemptible GC.

A typical GC process consists of a series of page read, data transfer, page write, and meta data update and erase operations as described in Figure 2. As illustrated in Figure 9, suppose that a write request arrives during a page read. As discussed in the previous subsection, FTL checks if the read command is completed by polling the status register. While polling the status register FTL also looks up the incoming request queue to check if any request comes during the on-going operation. If a request arrives, FTL issues a suspend command to stop the current read command and services the write command. Looking up the request queue does not incur an additional overhead because it occurs while polling the status register and time spent on polling never contributes to the performance.

TABLE III: Handling requests on the same logical page of the on-going command.

Active operation	Incoming operation	Action
Read	Read	Request is serviced by the on-going command.
	Write	Active read operation is discarded.
Write	Read	The request is serviced from the buffer used by the on-going command.
	Write	Data written by the on-going command are invalidated
Erase	Read	Not happen
	Write	Not happen

The incoming request may happen to be on the same logical page of the on-going command. Table III summarizes cases of conflicts. If the incoming request is a read on the same logical page of the on-going read command, the on-going read

command doesn't need to be suspended. Once the current read command finishes, data in the page buffer can be used for servicing the incoming request as well as for the following page write.

The incoming write request may be on the same *logical* page of the on-going read command. Then the data should be written to a different *physical* page. In this situation, the data read by the on-going read command are discarded because moving this page is not necessary any more.

Referring to Figure 8, suppose that the on-going read command and the incoming write request are on the same *logical* page and the logical page is mapped to *physical* page x before the read command is suspended. The on-going read command on page x is copying data from the NAND cell to page buffer A. When a write request comes on the same logical page, the on-going read command is suspended. The data to be written is stored in page buffer B and then a write command is issued to physical page y. After the write command finishes the meta data of page x and y should be updated as valid (V) to invalid (I) and empty (E) to valid (V), respectively as the mapping of the logical page is changed from physical page x to y. The data in page buffer A were supposed to be written by the following page write in the GC process. However, in this situation, data in page buffer A don't need to be written. The purpose of moving pages by GC is to move and invalidate all the valid pages in the victim block. In the case of page x, it is already invalidated by the incoming request and the up-to-date data are written to a different physical page. Therefore, page x doesn't need to be written by GC any more.

A request may come during the data transfer. Here, we also assume the data transfer is issued by the CPU. While moving data, the CPU also needs to look up the request queue because we assume an interrupt is not used. If the CPU looks up the queue frequently, it may shorten the response time of the incoming request, but it delays the completion time of the data transfer due to the overhead of the look-up.

When a request arrives during a page write, it can be serviced immediately by suspending the on-going write command. If the incoming request is a read request on the same logical page, it can be serviced directly from the page buffer without issuing a read command because the up-to-date data are stored in the page buffer which are being written to the NAND cell.

The incoming write may be on the same *logical* page of the on-going write command. Then the page written by the on-going write command is invalidated immediately after the command is completed. This situation is very similar to the example of Figure 9. Suppose that GC issues a write command to physical page x for moving a logical page. Before the write command is completed, a write request arrives on the same logical page. The incoming write request writes data to physical page y, which is the latest data. When resuming, the on-going page write to physical page x is completed but data in page x are stale. Therefore, physical page x is marked as invalid right after the on-going write finishes.

During meta data update the CPU needs to look up the request queue occasionally to service the incoming requests. How frequently the CPU should look up the queue also needs to be determined based on the trade-off between the response time of incoming requests and the overhead of the look-up.

If a request comes during an erase operation, it can be also serviced immediately by suspending the erase command. In this case, the incoming request cannot be on a page in the victim block that is being processed by the erase command. Before issuing the erase command, FTL should have moved all the valid pages, and the victim block contains only invalid pages. Therefore, there is no reason to read a page from the victim block. Also a page in that block cannot be written because the block is not erased yet.

D. Worst-Case Execution Time Analysis

While SSDs offer better average response time than HDDs, they often suffer from performance variability. From the view point of the file system, it looks non-deterministic when the request experiences long latency because it has no idea when GC delays the request. As will be demonstrated by the experiments, the proposed preemptible GC schemes attenuate the performance variability by reducing the worst-case response time. This subsection provides analysis on the worst-case response time to understand how the proposed GC schemes reduce the worst-case response time and performance variability.

To keep consistent with previous literatures [10], [34], we use the same terminology. The worst-case execution time (WCET) refers to the worst-case response time of incoming requests. Table IV summarizes the terminology used for WCET analysis.

TABLE IV: Terminology for WCET analysis.

Symbol	Definition
T_{er}	Time to erase a block
$T_{suspend}$	Time to suspend an on-going command
$U(e_r)$	Upper bound of time to read a page
$U(e_w)$	Upper bound of time to write a page

T_{er} denotes the time to erase a block. It corresponds to the time taken to complete an erase command on the NAND flash chip. $T_{suspend}$ means the time to suspend an on-going command. Since suspending an erase command takes 20μs [37], we assume suspending all the commands takes 20μs. $U(e_r)$ and $U(e_w)$ denote the upper bound of time to read or write a page. These values vary with how the FTL manages the meta data.

TABLE V: WCET comparison.

Technique	Worst-case execution (response) time
GFTL [10]	$T_{er} + max\{U(e_r), U(e_w)\}$
RFTL [36]	$max\{T_{er} + U(e_w), U(e_r)\}$
PGC	$T_{er} + max\{U(e_r), U(e_w)\}$
FPGC	$T_{suspend} + max\{U(e_r), U(e_w)\}$

Table V compares WCET of various techniques. It should be noted that WCET of PGC and FPGC is of *state 1* where all incoming requests are allowed to preempt GC. If the state is changed from 1 to 2 or 3 due to lack of free blocks, WCET would be increased. Since previous works [10], [34] don't take this pathological behavior into consideration, we only present WCET of state 1 in our comparison. WCET of PGC is the same with that of GFTL [10]. In PGC, on-going flash commands cannot be preempted. The longest command is the erase command. In the worst case, the request should wait

for the erase command to finish, which takes T_{er}. After it finishes, the request can be serviced which takes $U(e_r)$ or $U(e_w)$. Since the erase command cannot be merged with the request nor pipelined, the merging and pipelining cannot help to reduce WCET.

When FPGC is employed, any on-going command can be preempted, which takes $T_{suspend}$. Since $T_{suspend}$ is much smaller than T_{er}, WCET of FPGC is substantially shorter than PGC and other related techniques. PGC also offers WCET comparable to existing real-time FTLs [10], [34].

V. Experimental Results

A. Experimental Setup

We evaluate the performance of the PGC scheme using Microsoft Research's SSD simulator [1]. MSR SSD simulator is event-driven and based on the Disksim 4.0 [4] simulator. MSR SSD simulator has been used in several SSD related researches [32], [36]. In this paper, we simulated a NAND flash based SSD. SSD specific parameter values used in the simulator are given in Table VI.

TABLE VI: Parameters of SSD model.

Parameter	Value	Parameter	Value
Reserved free blocks	15 %	Blocks per plane	2048
Minimum free blocks	5 %	Pages per block	64
Cleaning policy	Greedy	Page size	4 KB
Flash chip elements	8	Page read latency	0.025 ms
Number of channels	8	Page write latency	0.200 ms
Planes per element	8	Block erase latency	1.5 ms

To conduct a fair performance evaluation of our proposed PGC algorithm we fill the entire SSD with valid data prior to collecting performance information. Filling the entire SSD ensures that GC is triggered as new write requests arrive during our experiments. Specifically, for GC, we use a greedy algorithm that is designed to minimize the overhead of GC. The greedy algorithm selects a victim block to be erased whose number of valid pages is minimal. The more valid pages there are in the victim block, the longer it takes for GC to complete as the GC process needs to move more pages.

Our preemptible GC algorithm can be applied to any existing GC schemes, such as idle-time or reactive. In the idle-time GC scheme, the GC process is triggered when there are no new incoming requests and all queued requests are already serviced. In the reactive scheme, GC is invoked based on the number of available free blocks, without regard to the incoming request status. If the number of available free blocks is less than the set threshold, then the GC process is triggered; otherwise, it continues servicing requests. The reactive GC scheme is the default in the MSR SSD simulator, and we use it as our baseline (non PGC) GC scheme. The lower bound of the threshold in our simulations is set as the 5% of available free blocks. Ongoing GC is never preempted in the baseline GC scheme in our simulations. MSR SSD simulator implements a multi-channel SSD, and GC operates per channel basis. In our experiments, even if one channel is busy for GC, any incoming requests to other channels can be serviced. The preemption occurs only if the incoming request is on the same channel where GC is running.

We use a mixture of real-world and synthetic traces to study the efficiency of our semi-preemptible garbage collection scheme. We use synthetic workloads with varying parameters such as request size, inter-arrival time of requests, read access probability, and sequentiality probability in access.[2] The default values of the parameters that we use in our experiments are shown in Table VII.

TABLE VII: Default parameters of synthetic workloads.

Parameter	Value
Request size	32 KB
Inter-arrival time	3 ms
Probability of sequential access	0.4
Probability of read access	0.4

An exponential distribution and a Poisson distribution are used for varying request sizes and inter-arrival times of requests. Those distributions are well used to cover a variety of scenarios of workload cases in particular for the distribution of request arrivals. We vary one parameter while other parameters are fixed.

We use four commercial I/O traces, whose characteristics are given in Table VIII. We use write dominant I/O traces from an OLTP application running at a financial institution made available by the Storage Performance Council (SPC), referred to as the Financial trace, and from Cello99, which is a disk access trace collected from a time-sharing server exhibiting significant writes which was running the HP-UX operating system at Hewlett-Packard Laboratories. We also examine two read-dominant workloads. Of these two, TPC-H is a disk I/O trace collected from an OLAP application examining large volumes of data to execute complex database queries. Finally, a mail server I/O trace referred as OpenMail is evaluated.

TABLE VIII: Characteristics of realistic workloads. Note that bursty write percentage denotes the amount of write requests with less than 1.5 ms of inter-arrival times.

Workload	Avg. Req. Size (KB)	Read (%)	Arrival Rate (IOP/s)	Bursty Write (%)
Financial [41]	7.09	18.92	47.19	0.14
Cello [38]	7.06	19.63	74.24	31.32
TPC-H [44]	31.62	91.80	172.73	11.29
OpenMail [17]	9.49	63.30	846.62	0.01

While the device service time captures the overhead of GC, it does not include queuing delays for pending requests. Additionally, using an average service time does not capture response time variances. In this study we utilize (i) the system service response time measured at the block device queue and (ii) the variance in response times. Our measurement captures the sum of the device service time and the additional time spent waiting for the device (queuing delay) to begin to service the request.

B. Performance Analysis of Semi-Preemptible GC

The following garbage collection schemes are evaluated in this subsection:

- **NPGC**: A non-preemptible garbage collection scheme.
- **PGC**: A semi-preemptible garbage collection scheme with both merging and pipelining enabled.

[2]If a request starts at the logical address immediately following the last address accessed by the previously generated request, we consider it a sequential request; Otherwise, we classify it as a random request.

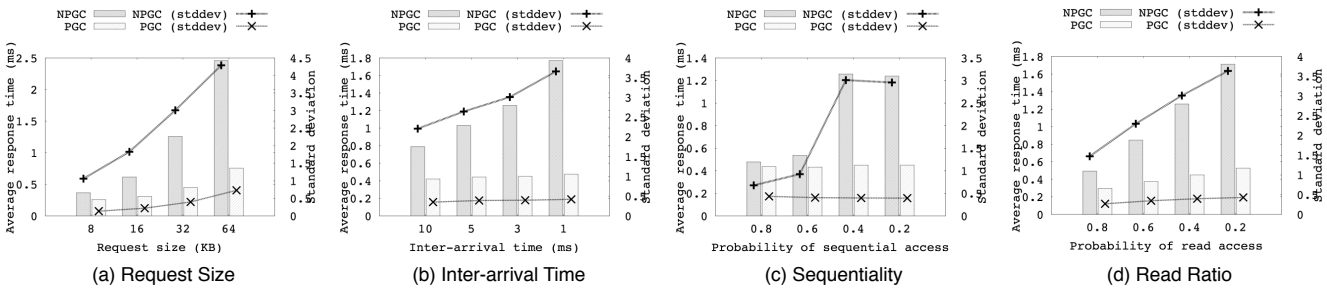


Fig. 10: Performance improvements of preemptible GC for synthetic workloads. Average response times and standard deviations are shown with different parameters of synthetic workloads.

1) Performance analysis for synthetic workloads: To evaluate the performance of PGC with various characteristics of input workloads, we start evaluating PGC with various synthetic workloads. GC may have to be performed while requests are arriving. Recall that GC is not preemptible in the baseline GC scheme and incoming requests during GC are delayed until the on-going GC process is complete. Figure 10 shows the performance improvements when enabling GC preemption.

a) Request size: Figure 10(a) shows the improvements of performance and variance by PGC for different request sizes In this experiment, we vary the request size as 8, 16, 32, and 64 KB. These values are chosen because the average request size of realistic workloads is between 7 and 31 KB, as given in Table VIII. For a small request size (8 KB) we see the improvement in response time by 29.44%. Furthermore, the variance of average response times decreases by 87.31%. As the request size increases, we see further improvements. For a large request (64 KB), the response time decreases by up to 69.21% while its variance decreases by 83.03%.

b) I/O arrival rate: Similar to the improvement with respect to varying request sizes, we also see an improvement with respect to varying the arrival rate of I/O requests. Typical response time of a request on a page is less than 1 ms without GC while it can be as high as 3-4ms when the page request is queued up due to GC. Based on this observation, we vary the inter-arrival time between 1 and 10 ms in our experiments. In Figure 10(b), it can be seen that PGC is minimally impacted by intense arrival rate. In contrast, the system response times and their variances for the baseline (NPGC) increase with respect to the request arrival rate.

c) Sequential access: Random workloads (where consecutive requests are not next to each other in terms of their access address) are known to be likely to increase the fragmentation of SSD, causing a GC overhead increase [21], [15]. We experiment with PGC and NPGC by varying the sequentiality of requests. Figure 10(c) illustrates the results. As can be seen, NPGC exhibits a substantial increase in system response time and its variance for a 60% sequential workload while PGC performance levels remain constant for all levels of sequentiality.

d) Write percentage: Writes are slower than reads in SSDs because flash page writes are slower than reads (recall unit access latency for reads and writes, 25us and 200us, respectively) and GC can incur further delays. In Figure 10(d), we see the improvement of PGC as the percentage of writes within the workload increases. Overall, we observe that PGC exhibits a marginal increase in response time and variance compared to the NPGC scheme. For example, PGC performance slows down by only 1.77 times for an increase of writes in workloads (from 80% to 20% of reads) while NPGC slows down by 3.46 times.

From the performance analysis with synthetic workloads, we can observe a firm trend that PGC improves the performance, regardless of workload characteristics, and has a beneficial impact on the performance when the workload is heavier (e.g., larger request size, shorter inter-arrival time, less sequentially and more write access).

2) Performance analysis for realistic server workloads: This sub-subsection evaluates the performance of PGC with realistic server workloads. Merging and pipelining techniques and the safeguard are evaluated individually. The following garbage collection schemes are added for the evaluation in this sub-subsection:

- **PGC+None**: A semi-preemptible garbage collection scheme without any optimization techniques.
- **PGC+Merge**: Only merging technique enabled PGC.
- **PGC+Pipeline**: Only pipelining technique enabled PGC.

Figure 11 presents the improvement of system response time and variance over time for realistic workloads. For write-dominant workloads, we see an improvement in average response time by 6.05% and 66.56% for Financial and Cello, respectively (refer to Figure 11(a)). Figure 11(b) shows a substantial improvement in the variance of response times. PGC reduces the performance variability by 49.82% and 83.30% for each of the workloads. In addition to the improvement in performance variance, we observe that PGC can further reduce the maximum response time of NPGC by 77.59% and 84.09% for Financial and Cello traces as illustrated in Figure 11(c).

For the OpenMail trace PGC does not show a significant improvement for performance and variance, as we expected for read-dominant traces. However, PGC reduces the maximum response time by 60.26%. Interestingly for TPC-H, although it is a read dominant trace, we observe a substantial improvement for performance and variance. TPC-H is a database application. The disk trace includes a phase of application run that inserts tables into a database, which is shown as a series of large write requests (around 128 KB) for database insert operations.

Moreover, we observe further improvement by the pipelining technique on PGC in the Figure 11.

Table IX shows how much the merging and pipelining contribute to the performance enhancement. The numbers shown

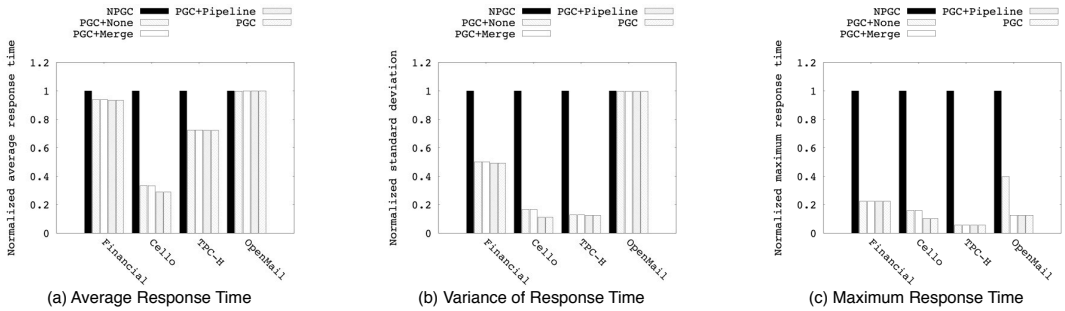


Fig. 11: Performance improvements of PGC and PGC+Pipelining for realistic server workloads.

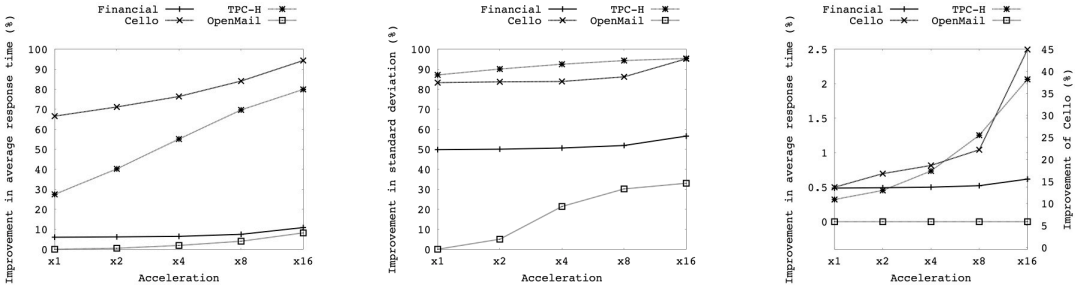


Fig. 12: Scalability tests by increasing the arrival rate of I/O requests.

in this table are the percentage of NAND flash commands affected by merging or pipelining among all flash commands issued by the incoming requests. Let N_w be the number of total write requests and N_r, the number of total read requests. The number of actual flash commands may not be the same because a request may span to multiple commands to multiple packages. Let's denote the number of write commands by C_w and that of read commands by C_r. Out of C_w commands, M_w commands are merged into commands issued by the on-going GC. Similarly, P_w commands are pipelined with commands of GC. Then, the percentage of write commands affected by merging is computed by $\frac{M_w}{C_w+C_r}$. The percentage of write commands affected by pipelining is $\frac{P_w}{C_w+C_r}$. Those of read commands are computed in the same way.

TABLE IX: Percentage of NAND flash commands affected by merging and pipelining.

Benchmark	Merging		Pipelining	
	Read	Write	Read	Write
Financial	0.10%	0.13%	7.29%	36.71%
Cello	0.01%	0.14%	10.46%	44.23%
TPC-H	0.01%	0.01%	8.82%	0.79%
OpenMail	0.00%	0.00%	0.00%	0.00%

It is shown in Table IX that the chance of merging is very low. Especially, the chance of merging and pipelining for OpenMail is less than 0.001%. However we can still see that a high reduction of maximum response time can be achieved for OpenMail by I/O merge technique in Figure 11, although the average performance is not improved significantly.

The chance of pipelining is higher than that of merging. For Cello, an improvement is observed in the average response time of PGC by 13.69% and its performance variance by 33.53%. Note that pipelining one command may not contribute to improving the performance because a request may span to multiple read or write commands.

Continuous GC preemption can cause starvation of free blocks. Thus, we develop a mechanism that can avoid a situation where an entire system becomes completely unserviceable because no free blocks are available. For this, we implement our PGC algorithm with a hard limit of available free blocks. Our algorithm now has two thresholds, one is for triggering the GC process and the other is for stopping preemption. Once the number of free blocks reaches T_{hard}, SSD stops GC preemption. A hard limit (T_{hard}) is set for a lower bound of the number of free blocks available in SSD.

To illustrate the effect of our extra threshold, we use an amplified Cello trace where the arrival rate of I/O requests are 16 times higher and the average request size of our test workload is about 300 KB. Cello is chosen because Cello is the most write-intensive workload among the four benchmarks, but with the original traces, we did not observe the shortage of free blocks incurred by preemption. To evaluate the impact of the safe guard, we had to amplify the trace artificially. In Figure 13(a), we see the situation where there are no free blocks left due to continuous GC preemption and the SSD is not available to service the I/O requests. It captures a zoomed-in region for 7 seconds of entire simulation run. The remaining free blocks indicate the ratio of the number of available free blocks over the minimum number of free blocks. The minimum number of free blocks corresponds to the soft threshold (T_{soft}) which is 5% of the total number of blocks as shown in Table VI. On the contrary, in Figure 13(b) and (c), we see that the SSD handles the starvation of free blocks in the SSD by adjusting T_{hard}. We see that the lower T_{hard} shows better response time while it exhausts more free blocks.

Since there exists a trade-off between the number of free blocks and response times, we evaluate the impact of performance in terms of response time according to T_{hard}. Figure 14 shows the cumulative distribution function of response time for

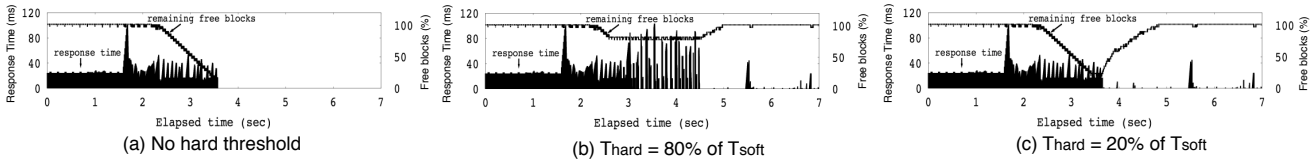


Fig. 13: Impact of hard threshold. The benchmark is Cello.

different T_{hard}. The average response times (in ms) are shown below each graph in the order of increasing the percentage of hard limit (T_{hard}). As we lower T_{hard}, we see overall response time improve. For example, we observe 18% improvement in average I/O response times when we lower T_{hard} from 80% to 20% of T_{soft}.

3) Performance Sensitivity Analysis: As shown in figures 12(a) and (b), with respect to increasing arrival rate, average response time and variance also improve. In particular, improvements in response times can be seen for write-dominant workloads (Financial and Cello) compared to read-dominant workloads in Figure 12(a). For TPC-H, we see a gradual improvement for the performance variability. Overall, we observe that PGC can increase the performance and improve the variance up to 90% for a 16 times more bursty workload (i.e. the I/O arrival rate is increased by 16 times). Figure 12(c) shows further improvements of the GC pipelining technique. In this figure, improvements in average response time for Cello can be clearly observed. Note that the scale for Cello is the right y-axis. For the other workloads, the benefit of the pipelining is not evident until the trace is accelerated significantly. The Financial and TPC-H exhibit a similar trend, but the OpenMail does not benefit from the pipelining because its chance is very low. However, we can still observe that the gaps of performance and variance are widened as the arrival rate of I/O requests increases. In other words, the GC pipelining technique makes PGC enabled SSDs robust enough to provide a sustained level of performance.

In addition to the greedy GC algorithm, we implemented two more GC algorithms to evaluate the performance of our proposed PGC for various real workloads. We implemented an Idle-based proactive GC algorithm where GC is triggered when an idle time is detected. For implementing idle time detection algorithm in workloads, we used a well-regarded heuristic on-line algorithm as in [13]. A wear-level aware GC algorithm has also been implemented [19]. Unlike the greedy GC algorithm, wear-level aware GC algorithm considers the wear-levels of blocks to avoid selecting a block that has experienced more erase operations than the average wear-out. The wear-level aware GC algorithm aims to distribute erase operations evenly across blocks.

Figure 15 shows the improvement of PGC against NPGC for various GC algorithms and various real workloads. We see that GC preemption works well regardless of GC algorithms. However, we see that the performance improvement of the idle-based algorithm is smaller than Figure 11. It is because idle-based GC algorithm can run GC in background, which doest not hurt the I/O service time. We also observe that Greedy-PGC outperforms Idle-NPGC for all the traces except for OpenMail. Even though GC runs during idle times, GC still has to run upon write requests when they come in a bursty manner. In case of OpenMail, the average response time and standard deviation of the idle-based GC algorithm is slightly higher than those of the baseline greedy GC algorithm. We speculate that running GC during idle times could make the operation sequence different, which affects the results, however this can be attributed to simulation artifact. Wear-aware GC algorithm does not show significant difference from the baseline of greedy GC algorithm.

From these experiments, we can observe that PGC reduces the response time and the variation regardless of GC algorithms. More importantly, it is shown that the PGC with a greedy GC algorithm (Greedy-PGC) that is triggered on demand will outperform the NPGC with a GC running during idle time (Idle-NPGC) in the background.

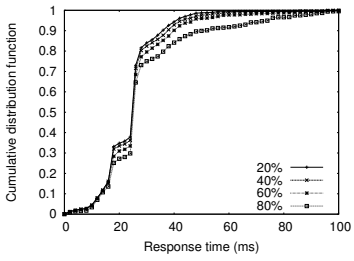
All the preceding experiments in this subsection were done without write-buffer. In this experiment, we study the impact of write-buffer on SSD. We considered STT-RAM based write-buffer. The read and write latency of STT-RAM is 20ns for both operations. STT-RAM has 10^{15} times of program/erase operation cycles, which is much higher than in NAND flash. Write-regulation technique that is a sort of selective write-buffering [23] can be employed if the lifetime of the STT-RAM buffer is seriously concerned. In our write-buffer implementation, data blocks are flushed into SSD whenever idle times in workloads are detected by flush operation.

Figure 16 shows the improvement of the average response time by using PGC compared against NPGC when an 1 MB write-buffer is employed. Compared with Figure 11(a), the performance improvement by using PGC is decreased, but PGC still improves the performance by 0.47%, 27.74%, 11.97% and 0.04% for Financial, Cello, TPC-H, and OpenMail, respectively. This experiments demonstrates that the proposed PGC improves the performance of write-intensive workloads even if a write-buffer is employed.

C. F-PGC Evaluation

After extensive evaluation of the semi-preemptible GC (PGC), we evaluate F-PGC and compare it with PGC. F-PGC has been evaluated with the same simulation environment described in Section V-A. We applied PGC and F-PGC to four realistic server workloads. We also implemented PGC+SE where suspend/resume commands are supported only for the erase operation. Note that suspend/resume commands can be operable with read, write and erase operations to implement F-PGC. The following garbage collection schemes are evaluated in this subsection:

- **PGC**: A semi-preemptible garbage collection scheme.
- **PGC+SE**: PGC with suspend/resume commands being supported only for the erase command.
- **F-PGC**: A fully-preemptible GC where suspend/resume



Avg. Resp. Times = {23.8, 24.4, 25.7, 29.1}

Fig. 14: Trade-off between response time and hard limit. The benchmark is Cello.

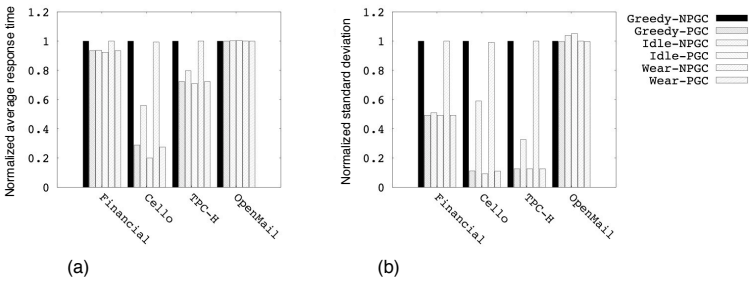


Fig. 15: Performance improvement of PGC for different GC algorithms.

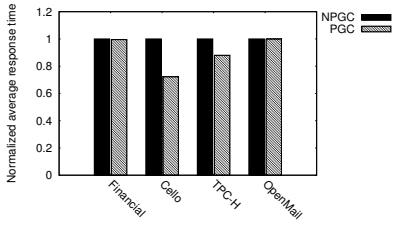


Fig. 16: Performance improvement of PGC over NPGC when an 1 MB write-buffer is employed.

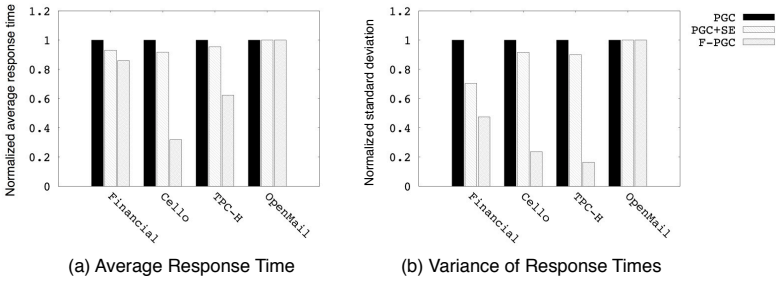


Fig. 17: Performance improvements of PGC+SE and F-PGC for realistic server workloads.

commands are supported for read, write and erase commands.

The suspend command takes up to 20μs [37] since a phase can last up to 20μs. Therefore, we assume the overhead of suspending all the operations as 20μs.

Figure 17 shows the normalized average response time and the normalized variance of response times. As shown in Figure 17(a) and (b), PGC+SE improves the average response time by up to 8.21% and the standard deviation by up to 29.63% compared to PGC. In case of F-PGC, it improves them by up to 68.13% and 83.59%, respectively. F-PGC shows significant improvements for Cello and TPC-H. Our conjecture is that Cello and TPC-H contain large amounts of bursty write requests, and F-PGC allows preemption on erase operations. Table VIII presents the percentages of write requests with less than 1.5ms of inter-arrival time for workloads. Note that 1.5ms is the block erase time on flash. Cello and TPCH have significantly higher percentages of bursty write requests than Financial and OpenMail. If an erase operation is not preemptible (which does in F-PGC), request during the erase operation will be delayed. Though Financial and Cello are write-dominant, Cello is bursty, while Financial is not bursty. Thus, F-PGC is not very effective for Financial. TPCH is a read-dominant workload, however, most of bursty write requests are gathered in the first part of the workload (less than 10% of total simulated time), and the remaining portion is mostly read requests, thus, F-PGC could significantly benefit from the first bursty write-dominant phase. OpenMail is read dominant, which has minimal impact on F-PGC.

The performance gain came mostly from preempting the erase and write operations. In our experiment, we allowed to preempt the read operation, but preempting the read operation did not have much impact on the performance because its chance for preemption was low and the latency of read was very short. Depending on the implementation, preempting the read operation may not be required.

VI. Related Work

To offer predictable performance, real-time FTLs [10], [34] adopt a similar GC scheme where incoming requests are serviced while GC is running. They will need additional free blocks in order to buffer incoming write requests to avoid interruptions. When a block is full, it is queued to be cleaned later by the GC process. If any write requests come to that block, they will be directed to a temporary buffer until the block is cleaned, then the pages in the buffer are moved to the original block, or their role is switched. The proposed PGC and FPGC do not need an additional buffer because they exploit the page buffer that already exists in the flash memory device (as explained in Section III-A).

Preemptible GC is discussed in [7] as a possible method to meet the constraints of a real-time system equipped with NAND flash. They proposed creation of a GC task for each real-time task so that the corresponding GC task can prepare enough free blocks in advance. In a real-time environment both GC tasks and real-time tasks need to be preemptible. However, since NAND flash operations can not be interrupted, these are defined as atomic operations. In contrast, our work provides a comprehensive study on the impact of the preemptible GC in an SSD environment (compared to real-time environment) and we emphasize optimizing performance by exploiting the internal parallelism of the NAND flash device (e.g. the multi-plane command and pipelining [32]).

Since it is well known that GC has significant adverse impact on the performance of SSD [10], [34], [16], [25], GC has attracted researchers' interest. Han [16] proposes using prediction to reduce the overhead of GC. An analytical model of the performance of GC [5] is developed to analyze the

impact of GC on the performance. Recently, Wu [39] reported that suspending the write and erase operations help to improve the performance. Although GC is not considered in his paper, his observation is in full agreement with ours. Kim [25] proposes a coordinated GC mechanism for an array of SSDs to improve performance degradation due to GC incoordination of individual SSDs.

In the HDD domain, semi-preemptible I/O has been evaluated [12] and its extension to RAID arrays also has been studied [12] by allowing preemption of on-going I/O operations to service a higher-priority request. To enable preemption, each HDD access operation (seek, rotation, and data transfer) is split into distinct operations. In-between these operations, a higher-priority I/O operation can be inserted. In the case of PGC, we allow preemption of GC to service any incoming request. We split GC operations into distinct operations and insert incoming requests in between them. In addition, we provide further optimization techniques while inserting requests.

VII. Concluding Remarks

Solid-state drives (SSDs) offer several advantages over HDDs: lower access latencies for random requests, lower power consumption, lack of noise, and higher robustness to vibrations and temperature. Although SSDs can offer better performance *on average* than HDDs in terms of I/O throughput (MB/s) or access latency, it often suffers from performance variability because of GC. From our empirical study, we observed that there are sudden throughput drops in commercially-off-the-shelf SSDs when increasing the percentage of writes in workloads. While GC is triggered to clean invalid pages to produce free space, incoming requests can be pending in the I/O queue, delaying their services until the GC finishes. This problem can become even more severe for bursty write-dominant workloads which can be observed in server-centric enterprise or HPC workloads.

To address this problem, we propose a semi-preemptible GC (PGC) that allows incoming requests to be serviced even before GC finishes by preempting on-going GC. We identified preemption points that incur negligible overhead during GC and found four states that prevent GC from starvation of I/O service that can occur due to excessive preemption. We enhance the performance even further by merging I/O requests with internal GC I/O requests and pipelining requests of the same type. We perform comprehensive experiments with synthetic and realistic traces. It is demonstrated by experiments that the proposed PGC can improves the average I/O response time by to up 66.56% and variance of response times by to up 83.30%. We applied PGC for accelerated workloads where inter-arrival time is shortened and evaluated with different GC schemes including idle-based proactive GC scheme and wear-aware selection algorithm. PGC exhibits significant performance improvement regardless of GC schemes for those workloads.

This paper also explores the feasibility of fully preemptible GC (F-PGC). Assuming that there is a NAND flash memory that supports suspend/resume commands for read, write and erase operations, we can implement F-PGC without incurring excessive overhead. Our evaluation result shows that F-PGC can further improve the average response time and the variation of response times by up to 14.57% and 52.48%, respectively, compared to PGC.

Acknowledgments

We would like to specially thank Doug Reitz for his detailed comments and proof-reading which helped us improve the quality of the manuscript. This research used resources of the Oak Ridge Leadership Computing Facility, located in the National Center for Computational Sciences at Oak Ridge National Laboratory, which is supported by the Office of Science of the Department of Energy under Contract DE-AC05-00OR22725. Also this work was also partially sponsored through Korea Ministry of Knowledge Economy grant (No. 10037244).

References

[1] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark Manasse, and Rina Panigrahy. Design tradeoffs for SSD performance. In *Proceedings of the Usenix Annual Technical Conference (USENIX ATC)*, June 2008.

[2] ARM. ARM security technology, 2009. http://infocenter.arm.com/.

[3] Joe. Brewer and Manzur. Gill. *Nonvolatile Memory Technologies with Emphasis on Flash (A Comprehensive Guide to Understanding and Using Flash Memory Devices)*. 2008.

[4] John S. Buch, Jiri Schindler, Steven W. Schlosser, Gregory R. Ganger, and et al. *The DiskSim Simulation Environment Version 4.0 Reference Manual*. http://www.pdl.cmu.edu/DiskSim/, 2008.

[5] Werner Bux and Ilias Iliadis. Performance of greedy garbage collection in flash-based solid-state drives. *Perform. Eval.*, 67(11):1172–1186, November 2010.

[6] Philip Carns, Robert Latham, Robert Ross, Kamil Iskra, Samuel Lang, and Katherine Riley. 24/7 characterization of petascale I/O workloads. In *Proceedings of the Workshop on Interfaces and Architectures for Scientific Data Storage*, 2009.

[7] Li-Pin Chang, Tei-Wei Kuo, and Shi-Wu Lo. Real-time garbage collection for flash-memory storage systems of real-time embedded systems. *ACM Transactions on Embedded Computing Systems*, 3(4):837–863, November 2004.

[8] Yuan-Hao Chang, Jen-Wei Hsieh, and Tei-Wei Kuo. Endurance enhancement of flash-memory storage systems: An efficient static wear leveling design. In *Proceedings of the 44th Annual Conference on Design Automation*, DAC '07, pages 212–217, New York, NY, USA, 2007. ACM.

[9] Feng Chen, David A. Koufaty, and Xiaodong Zhang. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *Proceedings of the eleventh International joint conference on Measurement and modeling of computer systems*, SIGMETRICS'09, pages 181–192, 2009.

[10] Siddharth Choudhuri and Tony Givargis. Deterministic service guarantees for nand flash using partial block cleaning. In *Proceedings of the 6th IEEE/ACM/IFIP International conference on Hardware/Software codesign and system synthesis*, CODES+ISSS'08, pages 19–24, New York, NY, USA, 2008. ACM.

[11] Tae-Sun Chung, Dong-Joo Park, Sangwon Park, Dong-Ho Lee, Sang-Won Lee, and Ha-Joo Song. System software for flash memory: A survey. In *Proceedings of the International Conference on Embedded and Ubiquitous Computing*, pages 394–404, August 2006.

[12] Zoran Dimitrijevi, Raju Rangaswami, and Edward Chang. Design and implementation of semi-preemptible IO. In *Proceedings of the USENIX Conference on File and Storage Technologies*, FAST'03, March 2003.

[13] Fred Douglis, P. Krishnan, and Brian Marsh. Thwarting the power-hungry disk. In *In Proceedings of the 1994 Winter USENIX Conference*, pages 293–306, 1994.

[14] Eran Gal and Sivan Toledo. Algorithms and data structures for flash memories. *ACM Computing Survey*, 37(2):138–163, 2005.

[15] Aayush Gupta, Youngjae Kim, and Bhuvan Urgaonkar. DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings. In *Proceeding of the 14th International conference on Architectural support for programming languages and operating systems*, ASPLOS'09, pages 229–240, 2009.

[16] Long-zhe Han, Yeonseung Ryu, Tae-sun Chung, Myungho Lee, and Sukwon Hong. An intelligent garbage collection algorithm for flash memory storages. In *Proceedings of the 6th International conference on Computational Science and Its Applications - Volume Part I*, ICCSA'06, pages 1019–1027, Berlin, Heidelberg, 2006. Springer-Verlag.

[17] Intel. Intel Xeon Processor X5570 8M Cache, 2.93 GHz, 6.40 GT/s Intel QPI. http://ark.intel.com/Product.aspx?id=37111.

[18] Intel. Intel X25-E Extreme 64GB SATA Solid-State Drive SLC. http://www.intel.com/design/flash/nand/extreme/index.htm.

[19] Dawoon Jung, Yoon-Hee Chae, Heeseung Jo, Jin-Soo Kim, and Joonwon Lee. A group-based wear-leveling algorithm for large-capacity flash memory storage systems. In *Proceedings of the 2007 International conference on Compilers, architecture, and synthesis for embedded systems*, CASES'07, pages 160–164, 2007.

[20] Jeong-Uk Kang, Heeseung Jo, Jin-Soo Kim, and Joonwon Lee. A superblock-based flash translation layer for NAND flash memory. In *Proceedings of the 6th ACM & IEEE International conference on Embedded software*, pages 161–170, 2006.

[21] Hyojun Kim and Seongjun Ahn. BPLRU: A buffer management scheme for improving random writes in flash storage. In *Proceedings of the USENIX Conference on File and Storage Technologies*, FAST'08, pages 1–14, February 2008.

[22] Youngjae Kim, Raghul Gunasekaran, Galen M. Shipman, David A. Dillow, Zhe Zhang, and Bradley W. Settlemyer. Workload characterization of a leadership class storage. In *Proceedings of the 5th Petascale Data Storage Workshop*, PDSW'10, November 2010.

[23] Youngjae Kim, Aayush Gupta, Bhuvan Urgaonkar, Piotr Berman, and Anand Sivasubramaniam. Hybridstore: A cost-efficient, high-performance storage system combining SSDs and HDDs. In *Proceedings of the IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, MASCOTS'11, July 2011.

[24] Youngjae Kim, Sudhanva Gurumurthi, and Anand Sivasubramaniam. Understanding the performance-temperature interactions in disk i/o of server workloads. In *Proceedings of the International Symposium on High-Performance Computer Architecture,* , HPCA'06, pages 179–189, Febuary 2006.

[25] Youngjae Kim, Sarp Oral, Galen M. Shipman, Junghee Lee, David A. Dillow, and Feiyi Wang. Harmonia: A globally coordinated garbage collector for arrays of solid-state drives. In *Proceedings of the 2011 IEEE 27th Symposium on Mass Storage Systems and Technologies*, MSST'11, pages 1–12, 2011.

[26] Junghee Lee, Youngjae Kim, Galen M. Shipman, Sarp Oral, Feiyi Wang, and Jongman Kim. A semi-preemptive garbage collector for solid state drives. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS'11, pages 12–21, April 2011.

[27] Sang-Won Lee, Dong-Joo Park, Tae-Sun Chung, Dong-Ho Lee, Sangwon Park, and Ha-Joo Song. A log buffer-based flash translation layer using fully-associative sector translation. *ACM Trans. Embed. Comput. Syst.*, 6(3):18, 2007.

[28] Sungjin Lee, Dongkun Shin, Young-Jin Kim, and Jihong Kim. LAST: locality-aware sector translation for NAND flash memory-based storage systems. *SIGOPS Oper. Syst. Rev.*, 42(6):36–42, 2008.

[29] H. Niijima. Design of a solid-state file using flash EEPROM. *IBM Journal of Research and Developement*, 39(5):531–545, 1995.

[30] ONFI. Open NAND flash interface specification. http://www.onfi.org/.

[31] Sarp Oral, Feiyi Wang, David A. Dillow, Galen M. Shipman, and Ross Miller. Efficient object storage journaling in a distributed parallel file system. In *Proceedings of the USENIX Conference on File and Storage Technologies*, FAST'10, February 2010.

[32] Seon-Yeong Park, Euiseong Seo, Ji-Yong Shin, Seungryoul Maeng, and Joonwon Lee. Exploiting internal parallelism of flash-based SSDs. *Computer Architecture Letters*, 9(1):9–12, January-June 2010.

[33] Steven L. Pratt and Dominique A. Heger. Workload dependent performance evaluation of the linux 2.6 i/o schedulers. In *Linux Symposium*, July 2004.

[34] Zhiwei Qin, Yi Wang, Duo Liu, and Zili Shao. Real-time flash translation layer for nand flash memory storage systems. In *Real-Time and Embedded Technology and Applications Symposium*, RTAS'12, pages 35–44, April 2012.

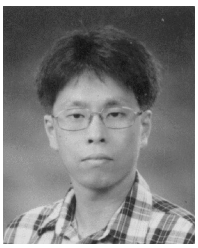
[35] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.*, 10(1):26–52, 1992.

[36] Ji-Yong Shin, Zeng-Lin Xia, Ning-Yi Xu, Rui Gao, Xiong-Fei Cai, Seungryoul Maeng, and Feng-Hsiung Hsu. FTL design exploration in reconfigurable high-performance SSD for server applications. In *Proceedings of the 23rd international conference on Supercomputing*, ICS'09, pages 338–349, 2009.

[37] Spansion. Am29BL162C data sheet. http://www.spansion.com/.

[38] Super Talent. Super Talent 128GB UltraDrive ME SATA-II 25 MLC. http://www.supertalent.com/products/ssd_detail.php?type=UltraDrive%20ME.

[39] G. Wu and X. He. Reducing ssd read latency via nand flash program and erase suspensions. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST'12, 2012.



Junghee Lee is currently a Ph.D. student at Georgia Institute of Technology. He received the B.S. and M.S. degrees in computer engineering from Seoul National University in 2000 and 2003, respectively. From 2003 to 2008, he was with Samsung Electronics, where he worked on electronic system level design of mobile system-on-chip. His research interests include architecture design of microprocessors, memory hierarchy, and storage systems for high performance computing and embedded systems.

Youngjae Kim is an I/O Systems Computational Scientist for the National Center for Computational Sciences at Oak Ridge National Laboratory. He received the B.S. degree in computer science from Sogang University, Korea in 2001, the M.S. degree from KAIST in 2003 and the Ph.D. degree in computer science and engineering from Pennsylvania State University in 2009. His research interests include operating systems, parallel I/O and file systems, storage systems, emerging storage technologies, and performance evaluation. He is currently an adjunct professor in the school of electrical and computer engineering at Georgia Institute of Technology.

Galen M. Shipman is the Data Systems Architect for the Computing and Computational Sciences Directorate at Oak Ridge National Laboratory. He is responsible for defining and maintaining an overarching strategy for data storage, data management, and data analysis spanning from research and development to integration, deployment and operations for high-performance and data-intensive computing initiatives at ORNL. Prior to joining ORNL, he was a technical staff member in the Advanced Computing Laboratory at Los Alamos National Laboratory. Mr. Shipman received his B.B.A. in finance in 1998 and a M.S. degree in computer science in 2005 from the University of New Mexico. His research interests include High Performance and Data Intensive Computing.

Sarp Oral is a Research Scientist at the National Center for Computational Sciences of Oak Ridge National Laboratory where he is a staff member of the Technology Integration Group. Dr. Oral holds a Ph.D. in computer engineering from University of Florida in 2003 and an M.Sc. in biomedical engineering from Cukurova University, Turkey in 1996. His research interests are performance evaluation, modeling, and benchmarking, parallel I/O and file systems, high-performance computing and networking, computer architecture, fault-tolerance, and storage technologies.

Jongman Kim is an assistant professor in the school of electrical and computer engineering at Georgia Institute of Technology. Dr. Kim received his B.S. degree from Seoul National University in electrical engineering in 1990. He received the M.S. degree in electrical engineering and his Ph.D. degree in computer science and engineering from Pennsylvania State University in 2001 and 2007, respectively. His research interests include hybrid multicore designs, Network-on-Chip, Massively Parallel Processing Architecture, and emerging memory systems. Before joining Pennsylvania State University, he had worked at LG Electronics and Neopoint Inc.